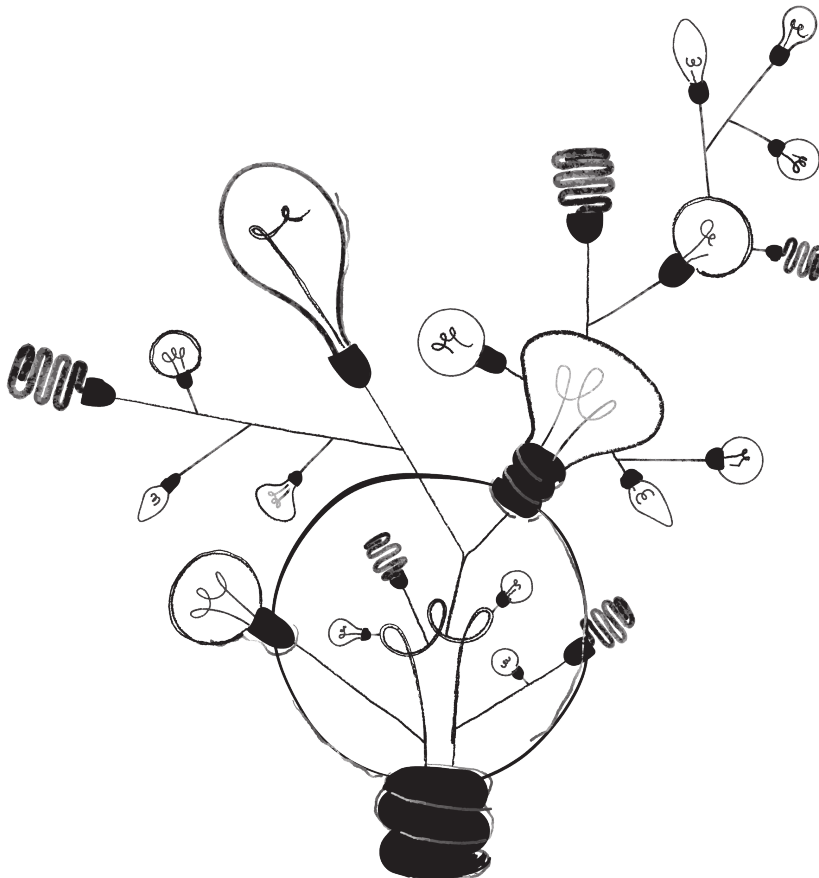


Esami commentati di Programmazione Funzionale

```
print "Hello World\n";
```

Emanuele Nardi

Compilato il 30 aprile 2019
v1.0.0



Questa pagina è stata lasciata vuota intenzionalmente

Introduzione

Lo scopo principale di questi appunti è quello di esaminare più da vicino gli esami di programmazione funzionale tenuti all'Università degli Studi di Trento. Questi appunti non sono completi, e la loro lettura non permette, da sola, di superare l'esame. La versione più recente si trova all'indirizzo:

github.com/emanuelenardi/latex-sml

Donazioni

Questa dispensa è in fase riscrittura, sudore e lacrime sono stati versati, per supportare l'autore in questo lungo e tortuoso viaggio effettua una piccola donazione [↗](#).

Ringraziamenti

Un grazie di cuore a:

- Alessio Gandelli che dice – “grazie per tutto”


Materiale

Puoi trovare una veloce introduzione a Standard ML su Learn X in Y minutes [↗](#).

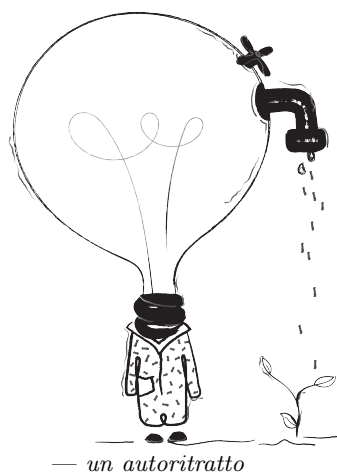
Ho prodotto una playlist di youtube [↗](#) che tratta gli argomenti del corso. Se trovi qualche video esplicativo e pensi che possa tornare utile ai tuoi compagni di corso, tramite questo link, puoi aggingerli direttamente alla playlist.

Per tutto il resto consulta la cartella Google Drive [↗](#) del corso triennale di Informatica.

Segnalazione di errori


Se hai trovato un errore ti prego di aprire un issue [↗](#) su  github.

Riguardo l'autore



Emanuele Nardi è uno studente di informatica all'Università degli Studi di Trento, Rappresentante degli studenti e co-fondatore di Speck&Tech.

Appassionatosi al design alle superiori, nei primi anni di università scopre il programma di impaginazione L^AT_EX con il quale inizia a prendere appunti durante le lezioni. Ha prodotto diverse dispense per il percorso di laurea triennale di informatica, il cui stato di completamento si può consultare su Awesome DISI [↗](#).

Tutti i suoi appunti sono disponibili sul suo profilo  github [↗](#).

Questa pagina è stata lasciata vuota intenzionalmente

Indice

I	Esami Pratici	7
	Giugno 2015	7
	Testo d'esame	7
	Guida alla soluzione	7
	Soluzione	8

Lista dei codici


1	Tipo di dato naturale	7
2	Funzione somma	7
4	Funzione somma alternativa . . .	7
6	Funzione prodotto	8
7	Esempio di esecuzione	9

Come leggere questa dispensa

Installazione

Il modo più semplice per imparare Standard ML è quella di interagire tramite un Read–Eval–Print Loop (REPL). Per installarlo ed eseguirlo in ambiente Linux:

```
$ sudo apt install polyml rlwrap      # installazione
$ alias poly='rlwrap poly'           # definizione di alias
$ poly                                # esecuzione della REPL
```

Altrimenti, se usi un sistema windows, puoi scaricare l'ultimo binario precompilato di Poly ML da  github. In questo caso ti consiglio di utilizzare una macchina virtuale per simulare lo stesso ambiente che ti si presenterà in sede d'esame.

Per eseguire il codice fornito è possibile copiarlo in un file che chiameremo `esegui.sml` e poi da terminale eseguirlo come uno script:

```
use "esegui.sml"
```

Trial and Error

Il Trial and Error è un modo comune e veramente efficace per imparare. Al posto di chiedere aiuto su ogni piccola cosa, qualche volta spendere un po' di tempo da soli (a volte ore e giorni) e provare a far andare qualcosa ti aiuterà ad imparare più velocemente.


Se provi qualcosa e ti dà un errore, studia quell'errore. Quindi prova a correggere il tuo codice. Quindi prova a eseguirlo di nuovo. Se ricevi ancora un errore, modifica ancora il tuo codice. Continua a provare e fallire finché il tuo codice non fallisce più. Imparerai molto in questo modo leggendo questa dispensa, leggendo gli errori e imparando cosa funziona e cosa no. Provare, fallire, provare, fallire, provare, provare, provare, fallire, fallire, avere successo!

Questo è quanto hanno imparato molti "pros". Ma non aver paura di chiedere aiuto, noi non mordiamo (duro). L'apprendimento richiede tempo, i professionisti che hai incontrato non hanno imparato a diventare maestri in poche ore o giorni.

Indentazione

L'indentazione è veramente importante! Il tuo codice funzionerà perfettamente senza, ma provocherà un grosso mal di testa a te e agli altri leggere il tuo codice.

Un breve spezzone di codice (25 linee o meno) probabilmente andrà bene senza indentazione, ma presto diventerà sciatto. È bene imparare ad indentare correttamente al più presto. L'indentazione non ha uno stile definito, ma è meglio mantenere tutto coerente.

Per approfondimenti vedi la voce "Indentation style"  su Wikipedia.

Chiedere aiuto

Prima di chiedere, prova a fare qualche ricerca tu stesso o prova a scrivere codice da solo. Se ciò non ha prodotto risultati che ti soddisfano, leggi di seguito.

- Non essere preoccupato di chiedere aiuto, anche le persone più intelligenti chiedono aiuto agli altri;
- Non essere preoccupato di mostrare quello che hai provato, anche se pensi che sia stupido (in particolare in questo caso, potresti aver trovato un modo più semplice di risolvere il problema);

- Posta qualsiasi cosa tu abbia provato;
- Fingi che chiunque tranne te sia un idiota e non sappia niente. Dai più informazioni possibili in modo da educare noi idioti su quello che stai cercando di fare;
- Aiutaci aiutati;
- Sii paziente, educato, aperto, gentile;
- Buon divertimento!

Parte I

Esami Pratici

Giugno 2015

Testo d'esame

Come noto, un numero naturale è esprimibile in base agli assiomi di Peano usando il seguente tipo di dato:

```
datatype naturale = zero | successivo of naturale;
```

Listing 1: Definizione di numero naturale

Usando tale tipo di dato, la somma fra numeri naturali è esprimibile come:

```
val rec somma = fn zero          => (fn n => n)
                  | successivo(a) => (fn n => successivo(somma a n));

val somma = fn: naturale -> naturale -> naturale
```

Listing 2: Definizione della funzione somma

Scrivere una funzione Standard ML, chiamata `prodotto`, che ha tipo `naturale -> naturale` e calcola il prodotto di due numeri naturali. Si noti che la funzione `prodotto` può usare la funzione `somma` nella sua implementazione.

Guida alla soluzione

Prendiamo confidenza con il tipo di dato definito:

```
> zero;
val it = zero: naturale

> successivo(successivo zero);
val it = successivo (successivo zero): naturale
```

Listing 3: Dichiarazione di numeri naturali

Commento sull'implementazione della funzione somma

La somma fra numeri naturali è esprimibile in due modi, equivalenti fra loro. Uno è quello illustrato dal professore, l'altro è il seguente:

```
val rec somma = fn zero          => (fn n => n)
                  | successivo(a) => (fn n => (somma a (successivo(n))));

val somma = fn: naturale -> naturale -> naturale
```

Listing 4: Definizione alternativa della funzione somma

Entrambe le definizioni della funzione `somma` sono corrette. Nella prima definizione il caso *successivo a* restituisce una funzione che mappa una variabile `n` nel successivo della somma di `a` con `n`, nella seconda definizione, invece, il caso *successivo a* restituisce una funzione che mappa una variabile `n` nella somma di `a` con il successivo di `n`.

Il funzionamento dell'esecuzione della funzione `somma` fra due numeri naturali è la seguente:

bisogna togliere un valore `successivo` al primo addendo affinché risulti pari al caso base (cioè zero).

Questo lo si fa o aggiungendo un valore `successivo` alla somma del primo addendo con il secondo (1^a implementazione) o sommando il primo addendo con il successivo del secondo addendo (2^a implementazione).

La funzione `somma`, quindi, non svolge nient'altro che una funzione di *wrapper* per la ricorsione la quale finisce nel momento in cui il valore associato alla variabile `a` raggiunge il caso base, ossia `zero`.

```
> somma (successivo zero) (successivo (successivo zero));  
val it = successivo (successivo (successivo zero)): naturale
```

Listing 5: Esecuzione di `somma`

Ovvero: la somma di 1 e 2, risulta 3.

Soluzione

Utilizziamo quindi la funzione `somma` nell'implementazione della funzione `prodotto`.

```
val rec prodotto = fn zero          => (fn b => zero)  
                  | successivo(a) => (fn b => (somma b (prodotto a b)));  
  
val prodotto = fn: naturale -> naturale -> naturale
```

Listing 6: Definizione della funzione `prodotto`

Esempio di esecuzione

Mostriamo un esempio di esecuzione della funzione `prodotto`:

```

(* definizione nuovo tipo di dato "naturale" *)
datatype naturale = zero | succ of naturale;

(* definizione della funzione ricorsiva "somma" *)
val rec somma = fn zero    => (fn n => n)
                  | succ(a) => (fn n => succ(somma a n));

(* definizione della funzione ricorsiva "prodotto" *)
val rec prodotto = fn zero    => (fn b => zero)
                    | succ(a) => (fn b => (somma b (prodotto a b)));

(* somma 1 1 *)
> somma (succ zero) (succ zero);
(* 2 *)
val it = succ (succ zero) : naturale

(* prodotto 2 2 *)
> prodotto (succ (succ zero)) (succ (succ zero));
(* 4 *)
val it = succ (succ (succ (succ zero))) : naturale

```

Listing 7: Esempio di esecuzione

Questa pagina è stata lasciata vuota intenzionalmente

Conclusione

Abbiamo raggiunto la fine del nostro viaggio, mio caro amico. Spero che tu abbia imparato qualcosa. Ma prima che te ne vada, considera di fare una piccola donazione ☞ .

Sei anche tu uno squattrinato studente fuori sede? Ecco cosa puoi fare:

- *Correggere e segnalare* gli errori contenuti del testo e/o negli esercizi;
- Condividere questa dispensa con un compagno di corso.